May 9, 1984
Volume III

"We don't need to test it. We only changed one line."

Everybody

Editor's Comments . . . . Ross Garmoe

Program Version Numbers . . . . Henry Burgess

Plotting on an Imagen Laser Printer . . . . Henry Burgess

Software Development Tools. . . . . Bob Matthews

Hardware Division Tools . . . . Marc Wilson

Tools Port to MS-DOS. . . . . Reuben Borman

EDITOR's COMMENTS - Ross Garmoe

As part of our effort, we have been porting the important Xenix tools to MS-DOS. In this effort, we have become convinced of the importance of a standard library which performs standard functions regardless of the machine or operating system being used. As part of the porting effort, we have assisted the C merge library group in the development of an MS-DOS library that simulates the standard C library of Xenix. When programs are developed using these standard libraries, porting is greatly simplified.

When new C language programs are developed, it is important that you use these standard libraries and not develop routines which have the same names as the standard routines but have different functions or effects. Routines which appear the same as the standard ones but operate differently make programs difficult to port. In addition, these "non-standard" library routines make debugging by others difficult because the programmer's mind-set is violated. If, for example, you develop a string copy routine that returns the pointer to the zero byte at the end of the string instead of a pointer to the beginning of the string, name it strzcpy instead of strcpy. Then, whenever another programmer encounters a reference to the routine, it is clear that something special is happening. One of the major reasons that C was selected by Microsoft as the standard development language is that it is easily portable across machines. To aid in this portability effort, it is important that we use standard libraries. Only in this way will we be able to quickly respond to new challenges and opportunities.

In the first issue of this journal, we announced the creation of the directory /usr/tools to hold the internal tools that were not part of the standard Xenix system. This directory now contains the programs used for 8086 cross development. We would like to extend the usefulness of /usr/tools by adding other routines that you have found useful. If you have any suggestions, please let me know via email (rossg). We hope developers will feel that their tools are useful for the company at large and submit the programs to /usr/tools. If you are developing programs for MS-DOS, then we will gladly accept these programs for distribution on the standard MS-DOS tools diskettes.

PROGRAM VERSION NUMBERS - Henry Burgess

It is a good idea to put version numbers in files.

Microsoft now has a standard for version numbers:

```
    "@(#)" filename version date"

    filename is the program/source file     name
    version is a string of the form         V1.0
    date is a string of the form            yy/mm/dd
```

You may place this string in the data area of a program or simply append it to the executable.

in DOS or XENIX:

```
    cat vers.txt   > > prog.exe
```

will do it. There will be a standard program for XENIX and DOS called 'what' that reports the version number(s) of files. The program keys on the starting sequence "@(#)" so the remaining portion of the string is free-form.

You should, however, follow the convention of filename-version-date.

PLOTTING ON AN IMAGEN LASER PRINTER - Henry Burgess

If you are on a XENIX system that has network access to an Imagen laser printer, then you may use the following command to produce a plot:

```
    graph   < datafile : plot -T4014 : ipr -Ltektronix
```

See the XENIX manual for details on the graph and plot commands.

SOFTWARE DEVELOPMENT TOOLS - Bob Matthews

In the past three years, the Productivity Software group has grown from a handful of people working on a single product to almost forty people working on a multitude of products. Fortunately, the value of providing these programmers with a reasonably powerful set of tools has been apparent enough to the powers that be to keep the wheels of tool development in motion. While far from perfect, the programming environment in the PS group is good enough to keep the development teams productive.

The typical PS programmer works with two or three others on a project. These people share a 68000 development system for source coding and compiling, then download to a target machine (typically a PC or Macintosh) for testing and debugging. Response time varies from barely tolerable to good, depending on how hard the other people on the project are working the host computer.

LANGUAGE

PS products are developed in a variant of the "C" language, and are compiled into P-code, which runs on an interpreter on the target machine. P-code is very dense and relocatable (hence swappable), which allows us to shoehorn a lot of functionality into limited memory. There is also little doubt that the portability afforded by the P-code architecture has been a key advantage in our race to provide quality and highly functional products for the Macintosh.

Our variant of C is very similar to standard C. Notable differences are that it supports arbitrary length identifiers, but doesn't support the passing of a variable number of arguments to procedures. The compiler originally only supported 16-bit operands, which was particularly limiting when trying to take advantage of the PC's larger address space. Extensions have since been added to manipulate 32-bit integers, and a method of accessing "far" locations has been implemented, but is far from simple to use. Work is currently underway to fully support "near" and "far" data pointers, which will provide a much more natural means of accessing large amounts of memory.

Additionally, the compiler will soon support a based pointer variant, called "lim", which can point to a data object of up to 64K in size. Code for "lim" pointers will be smaller and faster than for 32-bit "far" pointers.

Some static program analysis is provided by "lint". The usefulness of lint is limited by the extent to which the programmers have taken advantage of the extensions of our variant of C. Lint's seven significant character limitation on identifiers can be particularly annoying.

We have two locally developed programs to further aid in static program analysis. One, called "cross", provides the ability to get a full tree listing of which procedures call which procedures, or which procedures are called by which procedures, both up to a specifiable depth. This can be very helpful in "swap-tuning", the process of grouping related procedures into modules to keep the working set size to a minimum.

Another utility, called "xref" gives a global listing of objects and their types, and does a more in-depth analysis of type compatibility, including correctness of the number and types of arguments passed to procedures. This program catches some incompatibilities that lint passes over.

DEBUGGING

Debugging is usually done in one of three ways. If the program is character oriented, like Multiplan, it can be interpreted on the host machine itself, which avoids the process of downloading to the target machine. Much of our current work is on more graphically oriented products, such as Chart and Word, which are typically downloaded and debugged on the target machine.

Debugging on the PC is essentially the same as debugging on the host machine. In fact, the debugger code is the same for the two machines. When a breakpoint is reached, the interpreter stops handling the program's instructions, and runs the debugger instead.

Working in the confines of the Macintosh requires a more innovative approach. The debugger runs on the host machine, communicating with a specialized interpreter on the Macintosh, which runs the application code and answers requests from the debugger on the state of memory and registers. The programmer interacts with the application on the Macintosh, and with the debugger on his terminal.

The P-code debugger has most of the typical debugger features, and is in many ways similar to adb. You can set breakpoints, inspect data and disassemble code, single step, and run in "trace" mode. Program references are made relative to the beginning of the containing procedure ("GetChar + 3"), and data references are by global symbol name, or by absolute address. Local symbols are not supported, although you can ask for local variables by their position in the procedure's invocation block.

A unique and helpful feature of the debugger is the ability to set "watchpoints". A watchpoint is a directive to the debugger to break when a specified data location has been changed. The detection is done upon entry to and exit from procedures.

The debugger also does dynamic argument mismatch detection, which helps you verify that your procedures are at least being passed the correct number of arguments.

The debugger does have its limitations. It is certainly not a "source level" debugger, and some features commonly found in such debuggers are sorely missed. It would be very helpful for the debugger to interpret structure member references (to help decode the blocks of memory that make up the data structures of virtually any program). Although the debugger will disassemble code for your inspection, relating the code back to the source that produced it requires experience and/or the help of the "pximage" disassembly utility.

IMPROVING PERFORMANCE

At the tail end of the development cycle for a product, it usually becomes clear that some aspects need to be made faster. The first step in solving this problem is accomplished by using the measurement facilities of the P-code debugger. While at a breakpoint, you can tell the debugger to begin taking measurements for an interval. This produces a file that describes the activity of the program over that interval.

The ability to gather these measurements "on the fly", without having to compile special code into the program is a very convenient and powerful feature of the debugger. Unfortunately, generating the measurement file adds a tremendous amount of per-instruction overhead to the interpreter, which slows program execution dramatically for the duration of the measurement interval.

After the measurement file has been generated, a stand-alone utility is used to interpret the measurement file to produce reports that cover a wide range of detail. All of these measurements can be requested in terms of absolute instruction counts, or in terms of "weighted" instructions, where each instruction is weighted by how long it takes in real-time. This gives you an inexact but better idea of the true cost of the code. For example, floating point operations and procedure calls are heavily weighted.

Once the program has been optimized on an algorithmic level, there is often still a gap between real and desired performance. For applications designed to run on the 8086 or 68000 heavily used areas of the program can then be designated to run in native machine code, rather than interpretively. This gives a large performance enhancement, at the cost of higher code size. Combined with the profiling capability of the debugger, it makes for a powerful and efficient method of improving the overall performance of a product.

Studies abound that describe the tendency for programs to spend the majority of their time in a relatively small percentage of the code. With this selective machine coding capability, it is possible to get machine code performance out of those small portions, and greatly improve the overall performance of the product.

All that's required to have a section of the program run in native code is to bracket the desired group of source lines with "{{" and "}}". Further nesting of double braces causes the compiler to temporarily toggle back to P-code generation. This is a very useful feature, letting you enclose the body of a procedure in double braces to get machine code performance, and enclose the portions of that procedure which handle rarely encountered special cases in another level of double braces, to keep the overall size of the procedure to a relative minimum.

ASSEMBLING

The P-code interpreters and some other miscellaneous programs are often written in assembler. For the 8086, the assembler of choice seems to be Macro-86. Macro-86 runs on the 2020, on the 68000 systems, and on the PC itself. For 68000 work (primarily for the Macintosh), the "as" assembler under Xenix is used.

DOWNLOADING

We have downloaders and uploaders of the predictable variety, handling the tasks of moving files back and forth between 2020 and 8086 machines, between Xenix systems and 8086 machines, and between Xenix systems and the Macintosh. All are essentially derived from the old 2020 to CP/M-80 downloader.

EDITING

Short of maternal insults, making derisive comments about a programmer's choice of text editor is probably the most efficient way I know of to start a heated discussion. Fortunately, several adequate editors are available on the various host systems. E (the Rand editor) and Vi are the primary choices on the Xenix systems. Several PS programmers claim that EMACS is the ONLY adequate editor, and mourn its omission under Xenix. Z is commonly used on the 2020.

The limited editing which PS programmers do on the PC is usually done in Notepad, Word, or even Edlin. Z and Vi are available, although they don't seem to have caught on very well yet, probably because information on their availability has mostly been by word of mouth.

SOURCE MAINTENANCE

With several people working on a single product of considerable complexity, the task of making sure that each person has up to date versions of all the source files is a difficult task to manage by hand.

To help ease this situation, we have developed a program source library program, which manages the changing of shared source files. A master copy of each source file is kept in a special directory, and the participating programmers are given links to the master copy, which they can read, but not change. When the programmer wants to work on a module, he "checks out" that module from the system, which gives him a writable local copy, and keeps track of the fact that work on that module is under way.

If another programmer wants to work on the same module, the system informs him of the conflict, but can be commanded to check the module out anyway. When the original programmer "checks in" the modified module, the master copy is updated, giving the other programmers access to the latest version.

A helpful feature of the library system is the facility to keep people from checking modules in while compiles are in progress. This has helped us to avoid the mysterious bugs that crop up when a program is compiled while in an inconsistent state.

The library system keeps its records on a per-project basis. If work on the project spans more than one host machine, the file linkage upon which the library is based is no longer possible. This has proven to be a problem for several projects, but no reasonable solution has yet been implemented.

The library system also does not attempt to address the issue of multiple version maintenance or audit trailing, as "sccs" does. Perhaps a future implementation could assimilate the desirable aspects of such systems.

There's been a sharp increase recently in company-wide discussion on how to improve the development environment. Hopefully the preceding description of our environment here in PS will add some ideas to the discussion, as well as point out some local problem areas that may need to be addressed on a company-wide basis. Many of the tools described above have been used with some success outside of the PS group. Contact me or your favorite PS programmer if you are interested in getting software or more details.

HARDWARE DIVISION TOOLS - Marc Wilson

The people in the Hardware division work in a variety of environments. The tools used vary in quality, style and purpose. This description will be divided into several sections, namely, tools for hardware development, tools for source and documentation creation, tools for software development and miscellaneous tools. Following will be a brief discussion of future directions and an evaluation.

HARDWARE DEVELOPMENT, TESTING AND DEBUGGING

The circuit development is done by hand. The engineers specify chips, connections and other parts. The circuits are then usually wire-wrapped for testing. Once tested, a vendor is found to lay out the printed circuit board (PCB). A few PCBs are then manufactured and the PCBs are "stuffed". The hardware is then tested, usually with some diagnostics, sometimes with the actual software that is to be used with the hardware.

Currently, the hardware testing and debugging tools are fairly primitive (beside a fairly sophisticated serial data protocol analyzer and a logic analyzer). We occasionally use an In Circuit Emulator (ICE) or ICE-like device, but this usage is the exception. Mostly the hardware testing and debugging is done using the serial data analyzer, logic analyzer and oscilloscope and small programs written in BASIC or assembler.

The small P programs are written using BASIC interpreter, BASIC compiler, ASM86 and the debugger assembly facility. These programs are then either run directly (BASIC interpreter) or linked and run, ofter under Debug. The small programs for the Apple II are generated utilizing Applesoft, the Applesoft compiler and the Apple mini-assembler.

SOURCE AND DOCUMENTATION CREATION

Source files and documentation are created on a variety of machines using a variety of editors. The most popular editor is Z, running on Gonzo, XENIX machines and MS-DOS PCs. Also used are vi on XENIX machines; Word, PMate, and XYWrite on PCs; the Lisa monitor editor for Macintosh development; and Wordstar for SoftCard development under CP/M. Occasionally tools such as Grep, Trans, Dif, T96, mscomm and others are used. The COBOL source control facilities, Checkin and Checkout and the documentation preparation facilities nroff/troff are also utilized.

SOFTWARE TESTING AND DEBUGGING

Basic sources are executed using the BASIC interpreter on the MS-DOS PC and Apple II. They are also compiled by Microsoft's BASIC compilers and then linked using Microsoft linkers.

Developing assembly language programs is done in a wide variety of ways. SoftCard Z80 and 6502 programs are developed utilizing M80, L80, DDT and Apple's monitor. 8086 programs are developed on Gonzo using ASM86 and on machines using MASM and ASM87. They are downloaded using T96 and mscomm and linked using Microsoft's linker. The resulting EXE files are then executed, usually under Debug.

Developing the firmware to go into the ROMs that exist on our peripheral products is a little more difficult. We primarily use Avocet cross assemblers running on MS-DOS PCs. A data I/O prom burner is used to program the PROMs, usually downloaded from the MS-DOS PC. The processors include Z80, 6801, 6805, 8048 and Z8. The programs are debugged using the logic analyzer, serial data analyzer and oscilloscope.

Macintosh software is developed on the Lisa monitor system. It is written in Pascal or 68000 assembler and compiled, linked and downloaded to the Macintosh. On the Macintosh, the program is either run independently or under the Macintosh debugger. Various tools exist on the Lisa and the Macintosh which assist in the effort. The tools on Lisa include object file dumps and resource compiler (a resource is an object that a program on the Mac uses to effect its operation, e.g. code resource, window description resource, icon description resource and menu description resource). Tools used on the Macintosh include a resource mover/editor, setfiletype (tells the system whether a file is executable or is a document associated with another executable file) and a disk utility used for uploading/downloading and initializing disks.

MISCELLANEOUS

Email (both Oz on Gonzo and the XENIX mailer) is used for a variety of purposes, including reminder service (calendar.txt) and communication.

Make on XENIX systems and Do, Take and batch processing on Gonzo are used for often repeated sequences of operations (e.g. compiling and linking).

Phone program is used for retrieving phone numbers.

Z19 running on an MS-DOS PC is used in place of a terminal.

FUTURE DIRECTIONS AND EVALUATION

We are looking at Computer Aided Design machines to aid us in generating the hardware. We also would like to see a common environment for all of our development work. Switching from one system to another requires a fair amount of recall of details that distract from getting the job done. It would be extremely helpful if the firmware and hardware development (i.e. the assemblers, ICE, editor and possibly PCB layout) were all done in one environment.

The Xenix mailer is cumbersome when compared to Oz. A help facility similar to Tops-20 <esc> and ? would save a lot of time spent leafing through XENIX manuals for information on those seldom used commands.

Macintosh development is now supported by Apple on a Lisa in a fairly primitive fashion. Apple intends to improve the situation, but it is not clear how much commitment they have to get the job done. If C could support relocatable code and perhaps a few other Macintosh specific constraints and a symbolic debugger that ran on the Macintosh were developed, we would have a fairly complete Macintosh development system with Xenix.

TOOLS TO PORT TO MS-DOS - Reuben Borman

Xenix Utilities on MS-DOS -- Update

The following tools are now available:

ar	cat	cmp	diff	ed
egrep	fgrep	grep	hd	make
mkstr	more	od	sed	sort
split	tail	tar	touch	tr
vi37				

Note that tar can be used to transport sets of files in the form of an archive file between MS-DOS and Xenix.

Coming soon:

dc	bc	awk	lint	dd
pr	diff3			

Almost all tools are the System III versions and were ported with the C merge C compiler. To obtain copies, see Reuben Borman, X2250.

* * * *